بسم الله الرحمن الرحيم

لغة البرمجة جافا

Java Programming Language

الدرس الخامس :
البرمجة غرضية التوجه

<u>مفهوم الصف :</u>

إن برنامج java عبارة عن إنشاء مجموعة أغراض و استدعاء طرق مرتبطة بهذه الأغراض

لكي ننشأ غرض يجب أن يكون هناك توصيف له .. هذا التوصيف يعرف بالصف

فالصف هو توصيف لنوع معين من الأغراض . يمكن تعريف صفوف و استخدامها كما يمكن استخدام الصفوف المعرفة بشكل مسبق في مكتبات java

كما مر سابقا .. يتم إنشاء غرض باستخدام الكلمة المفتاحية new

قبل استخدام غرض ما يجب أن نقوم بتهيئته .. و تهيئة الغرض تتم باستخدام التابع الباني الذي يجب أن يكون معرف ضمن الصف .

اسم التابع الباني موافق لاسم الصف و السبب في ذلك يعود لأمرين

- **إذا لم يكن التابع الباني بنفس اسم الصف فربما يصبح الاسم متعارض مع اسم تابع آخر موجود داخل الصف**
- **و لأن عملية استدعاء التابع الباني تتم عن طريق المترجم لذا يجب أن يعرف ما اسم التابع الباني في كل صف**

<u>مثال :</u>

```
class Rock {
      Rock() {
            System.out.println("Creating Rock");
      }
}

public class SimpleConstructor {
      public static void main(String[] args) {
            for(int i = 0; i < 10; i++)
                  new Rock();
      }
}
```

و كأي تابع عادي .. التابع الباني يمكن أن يأخذ وسطاء لتحديد كيف سنبني الغرض

<u>مثال :</u>

```
class Rock2 {
      Rock2(int i) {
            System.out.println("Creating Rock number " + i);
      }
}
```

```
public class SimpleConstructor2 {
     public static void main(String[] args) {
          for(int i = 0; i < 10; i++)
               new Rock2(i);
     }
}
```

و لكن في المثال السابق تكون الطريقة الوحيدة لبناء الغرض هو بواسطة اتابع الباني بوسيط واحد

<u>التحميل الزائد للتوابع :</u>

أي مجموعة توابع تأخذ نفس الاسم و لكن تقوم بمهام مختلفة حسب احتياجاتنا ..

مثلا عندما نقول " اغسل القميص " و " اغسل السيارة " نلاحظ أنه فعل الغسل تم على القميص و السيارة

في جافا يمكن تحميل التابع الباني بشكل زائد حيث يكون لدينا أكثر من تابع باني كلها تملك نفس الاسم (اسم الصف) و لكن كل منها يقوم بتهيئة الغرض بأسلوب مختلف عن الآخر

كما يمكن تحميل أي من التوابع الأعضاء بشكل زائد ...

و يكون التحميل الزائد لتابع ما هو تابع جديد يحمل نفس اسم التابع القديم إلا أنه يختلف بعدد البارمترات و انماطها

و بالتالي نميز بين تابعين لهما نفس الاسم عن طريق قائمة البارمترات ..

لا يمكن التمييز عن طريق القيمة المعادة

<u>مثال :</u>

```
Void f() {}

Int f() {}
```

لو كان لدينا التابعين السابقين .. لهما نفس البارمترات و لكن يختلفان بالقيمة المعادة .. هنا لن يستطيع المترجم أن يميز بين التابعين في حال قمنا باستدعاء أي منهما

و لكن إذا قمنا بالاستدعاء التالي `int a=f()` ... في هذه الحالة سيعلم المترجم أي نسخة سيقوم بتنفيذها

و لكن إذا قمنا بالاستدعاء التالي `f()` .... هنا لن يستطيع المترجم التمييز ..

و تجنبا لهذه الحالات لا يمكن التمييز بين تابعين بالقيمة المعادة فقط ... و يؤدي القيام بذلك إلى ظهور خطأ في زمن الترجمة

<u>مثال :</u>

```
class Tree {

      int height;
      Tree() {
            prt("Planting a seedling");
            height = 0;
      }

      Tree(int i) {
            prt("Creating new Tree that is "
            + i + " feet tall");
            height = i;
      }

      void info() {
            prt("Tree is " + height
            + " feet tall");
      }

      void info(String s) {
            prt(s + ": Tree is "
            + height + " feet tall");
      }

      static void prt(String s) {
            System.out.println(s);
      }
}

public class Overloading {
      public static void main(String[] args) {
            for(int i = 0; i < 5; i++) {
                  Tree t = new Tree(i);
                  t.info();
                  t.info("overloaded method");
            }
            new Tree();
      }
}
```

تتم قراءة الكود و ملاحظة أين تم التحميل الزائد للتوابع

<u>الباني الافتراضي (الباني بدون وسطاء) :</u>

هو باني لا يأخذ بارمترات و يستخدم لبناء غرض خام .. لا يحمل أي قيمة ... أي بناء فقط لمجرد البناء

و إذا قمنا ببناء صف من دون أي باني فإن المترجم سيقوم بإنشاء باني افتراضي بشكل أوتوماتيكي

<u>مثال :</u>

```
class Bird {
      int i;
}

public class DefaultConstructor {
      public static void main(String[] args) {
            Bird nc = new Bird();
      }
}
```

نلاحظ أنه عندما عرفنا غرض من نوع `Bird` فإن المترجم قام باستدعاء الباني الافتراضي الغير موجود بشكل صريح

الآن .. إذا كان لدينا صف كمايلي :

```
class Bush {
      Bush(int i) {}
      Bush(double d) {}
}
```

و قمنا بكتابة السطر التالي `new Bush()` عندها يظهر لدينا خطأ و ذلك لأن المترجم لن يجد باني بدون وسطاء

<u>الكلمة المفتاحية `this` :</u>

لنفرض أننا داخل طريقة من طرق صف ما و نريد عنوان الغرض الحالي (الذي قام باستدعاء الطريقة) و لكن بما أنه لا نعلم اسم الغرض الحالي فإن المترجم سيمرره للطريقة بشكل مخفي و ذلك بواسطة الكلمة المفتاحية `this`

الكلمة المفتاحية `this` (تستخدم داخل الطرائق) تعبر عن عنوان الغرض الذي قام باستدعاء الطريقة و تستخدم تماما كأي غرض

<u>مثال :</u>

```
public class Leaf {
      int i = 0;

      Leaf increment() {
```

```
            i++;
            return this;
      }

      void print() {
            System.out.println("i = " + i);
      }

      public static void main(String[] args) {
            Leaf x = new Leaf();
            x.increment().increment().increment().print();
      }
}
```

بما أن الطريقة `increment()` تعيد عنوان الغرض الحالي لذلك يمكن اجراء عمليات استدعاء الطريقة `increment()` بشكل متتالي .

استدعاء البواني داخل البواني :

عندما يكون لدينا أكثر من باني ربما نقوم باستدعاء باني داخل باني و ذلك تجنبا لتكرار الكود و يتم ذلك باستخدام الكلمة المفتاحية `this`

معنى كلمة `this` هنا يختلف عن معناها سابقا ...

حيث أن كلمة `this` مع بارمترات تؤدي إلى استدعاء الباني الذي يطابق بارمتراته البارمترات الممررة لكلمة`this`

لا يمكن استدعاء `this` مع بارمترات أكثر من مرة في الباني الواحد

كما لا يمكن استدعاء `this` مع بارمترات في تابع غير باني

و استخدام آخر لـ `this` للتمييز بين بارمتر طريقة و حقل صف و ذلك عند تطابق الأسماء حيث للوصول لحقل صف نكتب

`this.name_of_member`

```
class a {

      int n;

      void p (int n) {

            this.n=n;

      }

}
```

مثال :

```java
public class Flower {
      int petalCount = 0;
      String s = new String("null");

      Flower(int petals) {
            petalCount = petals;
            System.out.println(
            "Constructor w/ int arg only, petalCount= "
            + petalCount);
      }

      Flower(String ss) {
            System.out.println(
            "Constructor w/ String arg only, s=" + ss);
            s = ss;
      }

      Flower(String s, int petals) {
            this(petals);
            //! this(s); // Can't call two!
            this.s = s; // Another use of "this"
            System.out.println("String & int args");
      }

      Flower() {
            this("hi", 47);
            System.out.println(
            "default constructor (no args)");
      }

      void print() {
            //! this(11); // Not inside non-constructor!
            System.out.println(
            "petalCount = " + petalCount + " s = "+ s);
      }

      public static void main(String[] args) {
            Flower x = new Flower();
            x.print();
      }
}
```

تهيأة الحقول الأعضاء :

في حال قمنا بتعريف متحول محلي داخل تابع و قمنا باستخدامه من دون إعطاءه قيمة بدائية فإن المترجم يعطينا خطأ في وقت الترجمة ...

مثال :

```
void f() {
      int i;
      i++;
}
```

أما بالنسبة لأعضاء صف ما ..

في حال كان الحقل من نوع أولي (primitive) و لم نقم بإعطاءه قيمة بدائية يقوم المترجم بإعطائه القيمة الابتدائية الافتراضية و إذا كان من نوع غرض و لم نقم بإعطاءه قيمة بدائية فإن القيمة البدائية هيnull . .

يمكن إعطاء قيم لأعضاء صف بعدة طرق ..

يمكن اعطاء قيمة لمتحول عضو في مكان تعريفه (سواء كان أولي أو غرض)

```
class A {
      boolean b = true;
      char c = 'x';

      Object o = new Object ();

}
```

كما يمكن إعطاء قيمة لمتحول عضو عن طريق تابع سواء كان مع بارمترات أو بدون بارمترات

```
class CInit {
      int i = f();
      //...
}
```

في حال كان التابع مع بارمترات و كان أحد البارمترات هو غرض من صف آخر فيجب أن نضمن أن هذا الغرض مهيأ ..

التعريف التالي صحيح :

```
class CInit {
      int i = f();
      int j = g(i);
      //...
}
```

بينما التعريف التالي خاطئ :

```
class CInit {
      int j = g(i);
```

```
	int i = f();
	//...
}
```

التهيئة باستخدام البواني :

عادة تستخدم البواني لإنجاز عملية التهيئة و لكن قبل الدخول لاستدعاء الباني تكون القيم الابتدائية للمتحولات الأعضاء هي القيم البدائية الافتراضية (كما ذكرنا سابقا) ...

مثلا :

```
class Counter {
	int i;
	Counter() { i = 7; }
	// . . .

}
```

تكون القيمة البدائية لـ i أولا 0 و بعدها تصبح 7

و نفس الشي بالنسبة لباقي أنواع المتحولات و الأغراض ..

مثال :

```
class Tag {
	Tag(int marker) {
		System.out.println("Tag(" + marker + ")");
	}
}

class Card {
	Tag t1 = new Tag(1); // Before constructor

	Card() {
		System.out.println("Card()");
		t3 = new Tag(33); // Reinitialize t3
	}

	Tag t2 = new Tag(2); // After constructor

	void f() {
		System.out.println("f()");
	}

	Tag t3 = new Tag(3);
}

public class OrderOfInitialization {
	public static void main(String[] args) {
		Card t = new Card();
```

```
        t.f(); // Shows that construction is done
    }
}
```

من خرج البرنامج السابق نلاحظ كيف أن القيم الابتدائية تعطى للمتحولات الأعضاء حتى قبل الدخول للتابع الباني

إعطاء القيم البدائية للمتحولات الأعضاء الستاتيكية :

يتم إعطاء القيم البدائية للمتحولات الأعضاء الستاتيكية تماما مثل المتحولات الأعضاء غير الستاتيكية .. إما في مكان تعريفها أو بواسطة تابع أو في الباني .

و لكن متى تتم هذه العملية ؟؟؟؟!!

المثال التالي يوضح الاجابة على هذا السؤال :

```
class Bowl {
    Bowl(int marker) {
        System.out.println("Bowl(" + marker + ")");
    }

    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

class Table {
    static Bowl b1 = new Bowl(1);

    Table() {
        System.out.println("Table()");
        b2.f(1);
    }

    void f2(int marker) {
        System.out.println("f2(" + marker + ")");
    }

    static Bowl b2 = new Bowl(2);
}

class Cupboard {
    Bowl b3 = new Bowl(3);
    static Bowl b4 = new Bowl(4);

    Cupboard() {
        System.out.println("Cupboard()");
        b4.f(2);
    }

    void f3(int marker) {
```

```java
            System.out.println("f3(" + marker + ")");
      }

      static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
      public static void main(String[] args) {
            System.out.println(
            "Creating new Cupboard() in main");
            new Cupboard();
            System.out.println(
            "Creating new Cupboard() in main");
            new Cupboard();
            t2.f2(1);
            t3.f3(1);
      }

      static Table t2 = new Table();
      static Cupboard t3 = new Cupboard();
}
```

من خرج البرنامج نلاحظ أن :

- عملية تهيئة المتحولات الستاتيكية تتم فقط عند الحاجة (عند تعريف غرض يملك أعضاء ستاتيكية)
- و تتم التهيئة لمرة واحدة فقط
- و تتم التهيئة قبل تهيئة الأعضاء غير الستاتيكية

يمكن أيضا تهيئة المتحولات الستاتيكية داخل كتلة static

<u>مثال :</u>

```java
class Cup {
      Cup(int marker) {
            System.out.println("Cup(" + marker + ")");
      }

      void f(int marker) {
            System.out.println("f(" + marker + ")");
      }
}

class Cups {
      static Cup c1;
      static Cup c2;

      static {
```

```
        c1 = new Cup(1);
        c2 = new Cup(2);
        }

        Cups() {
              System.out.println("Cups()");
        }
}

public class ExplicitStatic {
        public static void main(String[] args) {
              System.out.println("Inside main()");
              Cups.c1.f(99);
        }
}
```

و عملية تهيئة المتحولات الستاتيكية الخاصة بالصف السابق تتم عند الولوج إلى المتحول الستاتيكيc1 أو عند تعريف أول غرض من الصف Cups ...

و يمكن أيضا وضع كتلة لتهيئة المتحولات الغير ستاتيكية .. تماما كالمتحولات الستاتيكية و لكن تكون الكتلة غير مسماة .

تم بحمد الله